1 Lecture Outline

Reading: Chapter 8 Signal Processing Applications

- Plain reverberator (comb filter) (Section 8.2.1)
- Flanging and chorusing (Section 8.2.2)
- Digital sound field simulation (Section 8.2.3)
- Schroeder's reverberator

2 Plain Reverberation

We can add up an infinite number of successive echoes imitating reverberation using the following filter

$$y(n) = x(n) + ax(n-D) + a^2x(n-2D) + a^3x(n-3D) + \dots. \tag{1}$$

The impulse response and transfer function are given by

$$\begin{aligned} h(n) &= \delta(n) + a\delta(n-D) + a^2\delta(n-2D) + a^3\delta(n-3D) + \dots \\ &\updownarrow \\ H(z) &= 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D} + \dots \\ &= \frac{1}{1-az^{-D}}. \end{aligned} \tag{2}$$

The difference equation is then

$$y(n) = ay(n-D) + x(n). \tag{3}$$

The feedback delay causes the unit pulse to reverberate at multiples of D, that, is at $n = 0, D, 2D, \dots$.

Figure 1: Orfanidis p. 354 Figure 8.2.6 plain reverb.

The magnitude response and pole/zero pattern of the filter are given in Figure 8.2.7

3 plain.c

The following code implements the plain reverberator from the textbook.

plain.c

Figure 2: Orfanidis p. 354 Figure 8.2.7 IIR comb filter.

```c
#include "util.h"

float plain(int D, float *w, float **p, float a, float x)
{
    float y, sD;

    sD = tap(D, w, *p, D);  /* Dth tap delay output, sD = wD(n) */
    y = x + a * sD;         /* filter output, w0(n) = a wD(n) + x(n) = y(n) */
    **p = y;                /* blow away oldest state with newest state */
    cdelay(D, w, p);        /* update delay line, wk(n+1) = w{k-1}(n) */

    return y;   /* output sample */
}
```

4 C Implementation of Plain Reverberator

In this example, we will read an input text file containing interleaved right/left samples from a stereo audio signal. The right-channel samples will be reverberated using the plain reverberator while the left-channel samples will be passed unprocessed. The interleaved right/left output samples will be written to an output text file. The file I/O can be replaced with code which reads and writes samples to an audio device such as a soundcard.

main.c There is no change to main.c for this program.

util.h There is no change to util.h for this program.

user_data.h

```c
#ifndef USER_DATA
#define USER_DATA

void initialize_program();
void process_signal(float inputRight, float inputLeft, float *outputRight, float *outputLeft);

#define DELAY 1000              // desired feedback delay (in samples)

// extern variables are seen by *all* functions in *all* source files, i.e. global
// initialization of extern variables in initialize_program.c
extern float states[];          // buffer to store filter states
extern float *oldestStatePtr;   // pointer to oldest state in buffer
extern float a;                 // echo gain

#include "util.h"

#endif
```

initialize_program.c

```c
#include "user_data.h"

/*************************************/
/* Global variable initializations here */
```

```
5  /*************************************/
6  float states[DELAY+1]={0};       // storage for filter states initialized to zero
7  float *oldestStatePtr=states;    // pointer to oldest state in buffer
8  float a=0.8;                     // feedback gain
9
10 void initialize_program()
11 {
12 }
```

process_signal.c

```
1  #include "user_data.h"
2
3  void process_signal(float inputRight, float inputLeft, float *outputRight, float *outputLeft)
4  {
5      /*******************************/
6      /* Process right channel sample */
7      /*******************************/
8      *outputRight = plain(DELAY, states, &oldestStatePtr, a, inputRight);
9
10     /*******************************/
11     /* Process left channel sample */
12     /*******************************/
13     *outputLeft = inputLeft;
14 }
```

5 Flanging (8.2.3)

Interesting audio effects, such as flanging and chorusing, can be created by allowing the delay D to vary in time. For example, the echo difference equation may be replaced by

$$y(n) \quad = \quad x(n) + ax(n - d(n)) \tag{4}$$

A flanging effect can be created by periodically varying the delay $d(n)$ between 0 and 10 msec with a low frequency such as 1 Hz. For example, a delay varying sinusoidally between the limits $0 \leq d(n) \leq D$ will be

$$d(n) = \frac{D}{2}\left(1 - \cos(2\pi F_d n)\right) \tag{5}$$

where F_d is a low frequency, in units of [cycles/sample]. The realization is shown below.

Figure 3: Orfanidis p. 356 Figure 8.2.8 flanging effect.

The peaks of the frequency response of the resulting time-varying comb filter, will sweep up and down the frequency axis resulting in the characteristic whooshing type sound called flanging. The parameter a controls the depth of the notches.

Because the variable delay d can take non-integer values within its range $0 \leq d \leq D$, the implementation requires the calculation of the output $x(n-d)$ of a delay line at such non-integer values. This can be accomplished easily by truncation, rounding or linear interpolation. The tapi function computes the interpolated tap output of the delay line.

tapi.c p. 356

Figure 4: Orfanidis p. 358 Figure 8.2.9 flanged sinusoid signal.

```c
#include "util.h"

float tapi(int D, float *w, float *p, float d)
{
    int i, j;
    double si, sj;

    i = (int) d;            // interpolate between si and sj
    j = (i+1) % (D+1);      // if i = D, then j = 0; otherwise, j = i + 1

    si = tap(D, w, p, i);   // note, si(n) = x(n-i)
    sj = tap(D, w, p, j);   // note, sj(n) = x(n-j)

    return si + (d - i) * (sj - si);
}
```

main.c There is no change to main.c for this program.

util.h The only change to util.h is to include the tapi.c function prototype.

initialize_program.c

```c
float buffer[MAX_DELAY]={0};    // buffer to store samples initialized to zero
float *oldestSamplePtr=buffer;  // pointer to oldest sample in buffer
int n=0;                        // counter

void initialize_program(void)
{
}
```

user_data.h

```c
#ifndef USER_DATA
#define USER_DATA

void initialize_program(void);
void process_signal(float inputRight, float inputLeft, float *outputRight, float *outputLeft);

#define MAX_DELAY 21            // tapped delay line length
#define DELAY_LIMIT 20          // upper delay limit
#define FD 0.01                 // delay (low) frequency rads/sample
#define FS 100

// extern variables are seen by *all* functions in *all* source files, i.e. global
// initialization of extern variables in initialize_program.c
extern float buffer[];          // buffer to store samples
extern float *oldestSamplePtr;  // pointer to oldest sample in buffer
extern int n;                   // counter

#include "util.h"

#endif
```

process_signal.c

```c
#include "user_data.h"
#include <math.h>
```

```
3
4  void process_signal(float inputRight, float inputLeft, float *outputRight, float *outputLeft)
5  {
6      float d, s;
7
8      /*******************************/
9      /* Process right channel sample */
10     /*******************************/
11     *oldestSamplePtr = inputRight;  // insert newest sample into buffer
12
13     d = 0.5 * DELAY_LIMIT * (1.0 - cos(2 * M_PI * FD * n));
14
15     n += 1;
16     n = n % FS;
17
18     s = tapi((MAX_DELAY-1),buffer,oldestSamplePtr,d);    // get delay sample
19
20     *outputRight = 0.5 * (inputRight + s);           //  mix of new sample and delay sample
21     cdelay((MAX_DELAY-1), buffer, &oldestSamplePtr);     // back up pointer
22
23     /*******************************/
24     /* Process left channel sample */
25     /*******************************/
26     *outputLeft = inputLeft;
27 }
```

Audio Demo: flanging

6 Chorusing (8.2.3)

Chorusing imitates the effect of a group of musicians playing the same piece simultaneously. The musicians are more or less synchronized with each other, except for small variations in their strength and timing. These variations produce the chorus effect. A digital implementation of chorusing is shown in Fig. 8.2.10, which imitates a chorus of three musicians.

Figure 5: Orfanidis p. 358 Figure 8.2.10 chorusing effect.

The small variations in the time delays and amplitudes can be simulated by varying them slowly and randomly. A low-frequency random time delay $d(n)$ in the interval $0 \leq d(n) \leq D$ may be generated by

$$d(n) \quad = \quad D\,(0.5 + v(n)) \tag{6}$$

or, if the delay is to be restricted in the interval $D_1 \leq d(n) < D_2$

$$d(n) \quad = \quad D_1(D_2 - D_1)\,(0.5 + v(n))\,. \tag{7}$$

The signal $v(n)$ is a zero-mean low-frequency random signal varying between [-0.5, 0.5).

Audio Demo: chorusing

Figure 6: Orfanidis p. 359 Figure 8.2.11 chorusing of sinusoid signal.

7 Digital Sound Field Simulation (8.2.3)

The reverberation of a listening space is typically characterized by three distinct time-periods: 1) direct sound, 2) early reflections, and 3) late reflections.

1. The direct sound corresponds to the propagation delay of the acoustic wave taking the straight path between the source and listener
2. The early reflections correspond to the first few reflections (bounces) off the walls of the room, typically 1st order (one bounce) or 2nd order (two bounces)
3. The late reflections correspond to the waves which continue to bounce off the walls, in which their density increases and they disperse, arriving at the listener from all directions.

Figure 7: Orfanidis p. 362 Figure 8.2.15 reverberation impulse response of a listening space.

The sound quality of a concert hall depends on the details of its reverberation impulse response. Digital reverb processors attempt to simulate a typical reverberation impulse response. We note that the spectral peaks in the plain reverb filter

$$H(z) = \frac{1}{1-az^{-D}} \tag{8}$$

tend to accentuate those frequencies of the input signal that are near the peak frequencies.

7.1 All-Pass Reverberation or All-Pass Reverb

To prevent such spectral coloration of the input, we propose an allpass (flat magnitude response) version of the plain reverb filter

$$H(z) = \frac{-a+z^{-D}}{1-az^{-D}} \tag{9}$$

The I/O difference equation is given by

$$y(n) = ay(n-D) - ax(n) + x(n-D). \tag{10}$$

Figure 8: Orfanidis p. 364 Figure 8.2.17 Allpass reverberator in canonical and parallel form.

The frequency response is given by

$$H(\omega) = \frac{-a + e^{-j\omega D}}{1 - ae^{-j\omega D}} \tag{11}$$

and

$$|H(\omega)| = 1. \tag{12}$$

We can determine the impulse response as follows. First we write $H(z)$ in partial fraction form

$$H(z) = A + \frac{B}{1 - az^{-D}} \tag{13}$$

where $A = -1/a$ and $B = (1 - a^2)/a$. Expanding we have,

$$\begin{aligned} H(z) &= (A + B) + B\left(az^{-D} + a^2z^{-2D} + a^3z^{-3D}\dots\right) \\ &\updownarrow \\ h(n) &= (A + B)\delta(n) + Ba\delta(n - D) + Ba^2\delta(n - 2D) + \dots \end{aligned} \tag{14}$$

7.2 Schroeder's reverberator

The plain and allpass reverberators can be combined to form a more realistic reverb processor. Schroeder's reverberator is constructed as

Figure 9: Orfanidis p. 366 Figure 8.2.18 Schroeder's reverb processor.

With parameters as shown in the text, gives the following impulse response

Figure 10: Orfanidis p. 367 Figure 8.2.19 Impulse response of Schroeder's reverb processor.